RUNTIME VIRTUALIZATION AND DEVIRTUALIZATION OF I/O DEVICES BY A VIRTUAL MACHINE MONITOR

BACKGROUND

[0001] A virtual machine monitor ("VMM") creates an environment that allows multiple operating systems to run simultaneously on the same computer hardware. In such an environment, applications written for different operating systems (e.g., Windows, Linux) can be run simultaneously on the same hardware.

[0002] The VMM is loaded during bootup of the computer and receives control of the hardware at boot time. The VMM maintains hardware control until the computer is shut down.

[0003] Running an operating system on a VMM involves virtualizing one or more of the following types of hardware at bootup: memory, I/O devices, and CPU(s). CPUs can be virtualized in the traditional way: the hardware is configured to trap when an operating system ("OS") executes a privileged instruction, and the VMM simulates that instruction to maintain the illusion that the operating system has sole control of the hardware. To virtualize memory, the VMM typically manages memory translation in order to translate between the OS's use of physical memory, and the real "machine" memory present in hardware.

[0004] I/O virtualization typically involves two levels of device drivers, one in the OS, and one in the VMM. Drivers in each level provide their own interrupt handlers. In some systems, OS device drivers are designed with knowledge of the VMM drivers. In such systems, the OS device driver performs I/O by directly calling routines in the VMM driver. In other systems, the OS device driver is unaware of the VMM. In those other systems, the driver in the OS attempts to perform I/O as if that driver controlled the I/O device, but the VMM intercepts those attempts and instead performs the I/O on that driver's behalf, using the VMM device driver. I/O performed in this manner is called "emulated I/O." When interrupts from an I/O device occur, interrupt handlers in the VMM drivers get control first. They process the interrupt, and then, as needed, deliver the interrupt to the appropriate handler among the OS's device drivers.

[0005] Virtualizing the hardware adds overhead to the computer. For example, intercepting the I/O instructions performed by the OS drivers and using two levels of interrupt handlers slow the OS's normal I/O processing.

[0006] Since the VMM virtualizes the hardware from bootup to shutdown, overhead is incurred even when virtualization is not necessary (for example, when only a single OS instance is running on the hardware). Thus the VMM can add unnecessary overhead to the computer.

[0007] It would be desirable to reduce the unnecessary overhead.

SUMMARY

[0008] According to one aspect of the present invention, virtualization of an I/O device of a computer is commenced at runtime. According to another aspect of the present invention, devirtualization of the I/O device is commenced at runtime.

[0009] Other aspects and advantages of the present invention will become apparent from the following detailed description, taken in conjunction with the accompanying drawings, illustrating by way of example the principles of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] FIG. 1 is an illustration of hardware and software layers of a computer in accordance with an embodiment of the present invention.

[0011] FIG. 2 is an illustration of a method of commencing virtualization of an I/O device at runtime in accordance with an embodiment of the present invention.

[0012] FIG. 3 is an illustration of an exemplary state machine for an I/O device.

[0013] FIG. 4 is an illustration of a method of commencing I/O emulation in accordance with an embodiment of the present invention.

[0014] FIG. 5 is an illustration of a method of handling traps caused by I/O accesses in accordance with an embodiment of the present invention.

[0015] FIG. 6 is an illustration of performing virtualization and devirtualization in accordance with an embodiment of the present invention.

[0016] FIG. 7 is an illustration of performing runtime devirtualization of an I/O device in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION

[0017] As shown in the drawings for purposes of illustration, the present invention is embodied in a computer that can run a virtual machine monitor. The computer is not limited to any particular type. The computer could be, for example, a file server, web server, workstation, mainframe, personal computer, personal digital assistant (PDA), print server, or network appliance. The computer can be contained in a single box, or distributed among several boxes.

[0018] FIG. 1 shows different layers of an exemplary computer 100. The computer 100 has a raw hardware layer 110 and a software layer 111. The raw hardware layer 110 typically includes a central processing unit (CPU), memory and I/O devices. Exemplary I/O devices include, without limitation, network adapters, SCSI controllers, video cards, host bus adapters, and serial port adapters. The memory refers to the memory that is internal to the computer 100 (e.g., internal memory cache, main system memory) as opposed to external storage devices such as disk drives.

[0019] The software layer 111 includes an OS or OS instances 112, a VMM 114, and applications 116. The VMM 114 runs between the raw hardware layer 110 and the operating system instances 112. The VMM 114 allows one or more operating system instances 112 to run simultaneously. Applications 116 are run on the OS or operating system instances 112. During execution, the software for the software layer 111 can be stored in "articles" such as the memory; and during distribution, the software can be stored in articles such as external storage devices, removable media (e.g., optical discs), etc.

[0020] If the virtual and physical devices are compatible, the VMM can commence virtualization of the I/O device at runtime. Such virtualization at runtime is referred to as "runtime virtualization of I/O devices." The VMM can also cease the virtualization of these I/O devices at runtime. Such cessation is referred to as "runtime devirtualization of I/O devices." Runtime is the period of normal execution of the operating system after boot and before shutdown.

[0021] Physical and virtual devices are considered compatible if they both provide the same device interface. For example, two different serial line communications devices that both implement the interface of the Motorola 16550 Universal Asynchronous Receiver Transmitter (UART) would be compatible. Thus VMM and OS drivers are designed for the same interface.

[0022] Reference is made to FIG. 2, which illustrates a method of commencing virtualization of an I/O device at runtime. It is assumed that the VMM already has control over the CPU to perform this method. For example, the VMM is run in a more privileged mode than the OS and gets control on key hardware traps. In the alternative, the VMM has already virtualized the CPU but not the I/O (see FIG. 6)

[0023] First, interrupts are disabled (210). Next, interrupts from the I/O device are redirected from the OS to interrupt handlers in the VMM (212). The interrupts can be redirected by modifying the interrupt vector table. Next, the hardware is configured so the VMM intercepts accesses to the I/O device by the OS (214). This can be done by causing accesses to control registers in the I/O device to trap to a handler in the VMM in the manner described below. Next, interrupts are re-enabled (216). Finally, the VMM may commence device emulation (218). From that point forth, the VMM can emulate I/O normally by reconstructing high-level I/Os, and performing those I/Os using its own driver for the device.

[0024] The VMM can use a state machine to facilitate the emulation. An exemplary state machine for an idealized Ethernet card is illustrated in Figure 3. The exemplary state machine has four states. The VMM can initialize the state machine for the Ethernet card to the start state (state 1) and update its state whenever the OS attempts to access a control status register (CSR) of the Ethernet card. The state machine allows the VMM to reconstruct "high-level I/Os" (e.g., send message) from a sequence of "low-level I/Os" (i.e. the individual CSR accesses needed to carry out the high-level I/Os). The sequence of low-level I/Os performed to complete a single high-level I/O is referred to as an "I/O sequence". When the state machine reaches an "end state" (state 4), the VMM performs the reconstructed, high-level I/O using its own driver for the Ethernet card.

[0025] Ideally, the I/O processing should be paused so that virtualization takes place between I/O sequences. The I/O processing can be paused, for example, by the use of a kernel module in the OS, an application that issues a special IOCTL command to the driver within the OS instance, or some other technique that can pause I/O processing by the OS device driver between high-level I/O accesses.

[0026] However, because the VMM commences emulation in the middle of the I/O device operation, the emulation could commence in the middle of an I/O sequence. In this case, the VMM watches the OS's I/O accesses in order to determine when the current I/O sequence ends. Once the VMM has identified an interval between I/O sequences, the VMM can put the state machine for the emulated device in a start state, and commence emulation. The VMM can watch more than one complete I/O sequence to accurately determine an I/O sequence boundary.

[0027] Reference is made to FIG. 4, which provides an example of how the VMM can use the state machine to determine the interval between I/O sequences. First, the VMM initializes the state machine to the start state (410). Next, when the OS performs a low-level I/O access to the I/O device, that access traps to the VMM (412). The VMM then allows the trapping access to proceed (414) by emulating the instruction, single stepping the machine, etc. Next, the VMM checks whether there is a valid transition in the state machine from the current state for the trapped operation (416). If there is not, the VMM returns the state machine to the start state and awaits another trap (410). If there is a valid transition for the trapped operation, the VMM updates the state of the state machine as usual and checks whether the state machine has reached an end state (418). If an end state has not been reached, the VMM waits for another I/O operation on this device by the OS. If the state machine has reached an end state, the VMM resets the state machine to the start state (420). From that point on, the VMM can emulate I/O to the I/O device in a conventional manner by trapping future low-level I/O accesses by the OS (preventing those accesses from reaching the actual device), reconstructing high-level I/O accesses, and performing the high-level I/O accesses with its own driver.

[0028] The OS may issue an I/O access to the I/O device before virtualization, but that access later causes an interrupt to be delivered to the VMM after virtualization. From the moment interrupts are re-enabled (step 216 in Figure 2), the VMM device driver for the virtualized I/O device handles interrupts specially. The VMM ensures that the OS is not stuck waiting for an interrupt that gets delivered to the VMM driver instead, and ensures that the VMM driver is not confused by an unexpected interrupt. This special handling can take several forms. For simple devices, the VMM driver can simply drop interrupts it wasn't waiting for. Alternatively, the VMM driver can forward some interrupts to the corresponding OS driver (for example, by invoking its handler for the interrupt). When the VMM driver first starts up (at the end of step 218 in Figure 2), it may be able to query the state of the hardware to determine whether to expect any interrupts destined for the OS's driver, and forward them when they occur.

[0029] In certain cases, the VMM can pause new I/O processing by the OS long enough to allow any I/O activity initiated by the OS to finish generating interrupts, before allowing the driver in the VMM to take over. For example, between steps 216 and 218 of Figure 2, the VMM could trap and emulate accesses to the I/O device's CSRs as needed to tell the OS the device is in a "busy" state and cannot accept new I/Os. Alternatively, if the OS driver has sufficient support, an application may be used to issue an appropriate IOCTL command to pause the OS driver. A kernel module in the OS may also be able to grab a lock associated with the I/O device, or in some other manner prevent the OS from initiating a new I/O access to the I/O device.

[0030] In most cases, the VMM can stop handling interrupts specially once it has started issuing I/Os to the I/O device and receiving its own interrupts from the I/O device.

[0031] An example of intercepting I/O accesses and performing emulation will now be provided. The I/O device in this example is an Ethernet card.

[0032] In this example, the VMM uses memory management to trap I/O accesses. The VMM maintains control over address translation. Modern microprocessors provide the ability to translate between the "virtual" addresses used by applications and the OS, and the "physical" addresses used by the hardware. When the OS or application accesses a virtual address, that address is translated into a physical address to allow the access to reach the real memory or I/O device present in hardware. This translation is typically managed by software (e.g. the OS, the VMM, the firmware, etc.), but performed on the fly by the CPU.

[0033] Virtual and physical memory are divided up into small chunks of memory called pages. The mapping of virtual pages to physical pages is typically represented by "page tables". Each page table contains a list of page table entries ("PTEs"), and each PTE typically maps one page of virtual address space to one physical page and defines its permissions (read, write, execute, etc.) for that page (on some architectures one PTE can map more than one page). As the OS and application access virtual addresses, those accesses are typically translated into physical accesses by a Translation Lookaside Buffer ("TLB") within the CPU. When a virtual address is accessed for which there is not a valid translation in the TLB, the appropriate PTE is read from the current page table, and then loaded into the TLB. When running on a VMM, this fill operation may be performed by a TLB miss handler in the VMM (on some platforms and architectures), or it may be performed by firmware (in other architectures), or the PALcode (in the HP Alpha Architecture), or it may be performed by the hardware (in still other architectures).

[0034] By managing the CPU's virtual memory translation, the VMM can cause the CPU to trap to the VMM when the OS accesses addresses associated with the I/O device's CSR. How the VMM causes these traps depends upon how translations for memory access are inserted. For example, if the VMM handles TLB misses, it can refuse to load the TLB with a PTE for the page containing that virtual address, or it can install a PTE for that page with permissions such that the desired accesses will fault. When the desired TLB miss trap or protection trap occurs, the VMM handler for that event can check to see whether the faulting address is one of interest, then handle that access specially (for example, by emulating that access, modifying the result of that access, or translating that access to a different address).

[0035] If the VMM is not the party that handles TLB fills, the VMM can still cause particular virtual addresses to trap by configuring the CPU to use page tables defined by the VMM. By examining the OS page tables, the VMM can populate its page tables with the OS's translations as desired, but can also modify those translations as needed to cause the desired memory accesses to trap (e.g. by modifying the permissions for a particular page). The VMM can also add translations to its page tables to map or trap addresses beyond those the OS maps.

[0036] How the VMM traps the I/O accesses by the OS also depends on whether those accesses are to physical or virtual addresses. On many systems, the OS accesses I/O devices at physical addresses. Accesses to physical addresses normally bypass the TLB, making it difficult to trap them. However, most modern architectures have the ability to disable the direct addressability of physical memory and I/O space. By setting the appropriate bits in the CPU, the VMM can configure the CPU to treat accesses to physical I/O addresses as normal virtual accesses. After that, the VMM can manage those accesses in the manner described above.

[0037] Reference is now made to FIG. 5. To trap the OS's I/O accesses to addresses the OS has mapped into virtual memory, the VMM examines, and possibly modifies, the virtual-to-physical mappings set up by the OS. The VMM can do so either when the OS modifies its page tables (by configuring hardware to trap when those modifications take place), or upon a TLB miss trap for the address in question (if the VMM handles TLB fills). When either type of trap occurs (510), the VMM inspects the proposed translation by the OS (512), for example, checking to see whether the OS maps a virtual page to a physical page in I/O space containing an I/O device that the OS instance is allowed to use. The VMM is free to reject the translation (514), possibly emulating some kind of protection fault. The VMM is also free to modify the proposed translation (516), by changing, for example, the protections associated with that mapping so that future accesses will cause a protection fault, or mapping that virtual page to a different physical page than the OS proposed. The

VMM may also accept the proposed translation (518). After modifying or accepting the translation, the VMM allows virtual memory translation (VMT) to be performed using that translation (520), possibly by putting the translation into the TLB, or by storing it as a valid translation in the VMM's page tables.

[0038] Reference is made once again to Figure 3, which shows the state machine for an idealized Ethernet card. The state machine for this idealized Ethernet device is simpler than the one that would be used for a real Ethernet card. Operating systems (or their device drivers) typically perform I/O by setting up the I/O in memory, then instructing the device to perform the actual I/O operation. For example, to send a message using the Ethernet card, the OS might assemble the message in memory, then perform three writes to the CSRs on the attached Ethernet card. The first write to CSR "A" tells the card the location in memory of the outgoing message. The second write, to CSR "B", tells the card the number of bytes in the message. The third write, to CSR "C", informs the card of the Ethernet address of the destination host. The Ethernet card, once it has received these three writes, has sufficient information to complete the OS's request, so it sends the message. The OS may also have to read control registers on the card, for example, to determine whether the card is buffering a received message, or whether it is ready to accept a new message to send.

[0039] In virtualizing an OS's access to an I/O device, the VMM observes the operations in an I/O sequence. Based on those accesses, the VMM reconstructs a high-level I/O, and then performs that high-level I/O on behalf of the OS. The VMM uses its own device driver to perform the high-level I/O operation.

[0040] The VMM may perform the high-level operation as follows. The VMM configures the hardware to trap when the OS reads or writes to the address of each CSR associated with that device. Each time the OS traps to the VMM while accessing the device, the VMM changes the state of its state machine for that device, then prevents that device load or store instruction from actually reaching the I/O device.

[0041] The state machine starts in State 1, the "start" state. When the OS attempts to write the address of the message buffer to CSR A, the VMM traps that access and updates the state machine to State 2 and files away the buffer's address in memory. When the VMM traps the second I/O write (for writing the number of bytes in the message to CSR B), the VMM updates the state machine to State 3, and files away the message length. When the OS attempts to write the destination Ethernet address to CSR C, and that access traps to the VMM, the VMM updates the state machine to State 4, and remembers that Ethernet address. Since state 4 is an "end" state for this state machine, the VMM now has sufficient information to perform the OS's high-level I/O (message send) using its own Ethernet driver. After initiating that I/O with its driver, the VMM can reset the state machine to the start state to prepare for another I/O sequence. State machines may have more than one start state, and more than one end state.

[0042] If the VMM observes an access to the Ethernet card other than the ones allowed by the legal state transitions in the state machine, the VMM handles the error appropriately, perhaps by dropping the I/O, or by emulating a "target abort" error condition.

[0043] To correctly emulate the Ethernet card, the VMM may also emulate interrupts from the Ethernet card. The VMM device driver can handle interrupts from the Ethernet card. After it has processed an interrupt, the VMM may choose to deliver an interrupt to the OS by executing a "jump" instruction to the appropriate interrupt handler in the OS.

[0044] The type of the virtualized device used by the OS does not have to match the type of device installed in the computer. For example, the VMM could tell the OS during the OS's resource discovery that the hardware contains a 3Com Ethernet card. The VMM could then employ a state machine compatible with that card to reconstruct the OS's network I/Os, but then perform the actual network I/Os using an Ethernet card having a device interface different from the 3Com device, using its own driver for the Intel card. However, the virtual and physical devices are compatible..

[0045] Reference is now made to FIG. 6. The VMM can also devirtualize the I/O device at runtime. The devirtualization includes stopping I/O device emulation at runtime.

[0046] The devirtualization can be performed after runtime virtualization, or it can be performed after virtualization at VMM bootup. For example, the I/O device can be virtualized at bootup of the VMM (610). If not used immediately, the I/O device can be immediately devirtualized (612) to reduce any unnecessary overhead associated with the virtualized I/O device. When it becomes necessary to virtualize the I/O device, the runtime virtualization is performed (614). When a virtualized I/O device is no longer needed, the I/O device is devirtualized again (616).

[0047] The decision to perform runtime virtualization and devirtualization can be made by a system administrator. The system administrator informs the VMM of the decision (e.g., using a GUI).

[0048] Reference is made to FIG. 7, which illustrates an exemplary method of devirtualizing an I/O device at runtime. The VMM postpones devirtualization until the state machine for the I/O device enters the start state, and blocks the OS from commencing a new I/O sequence (710). There are many possible ways the VMM can block the OS from commencing a new I/O sequence. As a first example, an application may be used to issue an appropriate IOCTL command to pause the driver, if the driver has the needed support. As a second example, a kernel module in the OS may grab a lock associated with the device in order to prevent new I/O sequence from being started. As a third example, the VMM may emulate the device as being either "busy" or in a "device not ready" state. As a fourth example, thee VMM may trap the first operation in a new I/O sequence, and simply block the entire OS. As a fifth example (an alternative to pausing I/O processing for the device), the VMM may trap and log the OS's I/O operations during devirtualization, and simply replay the log to the I/O device once devirtualization is finished.

[0049] Next, the VMM waits for any I/Os to complete that were enqueued by the VMM device driver (712). The VMM driver then waits for any expected interrupts and clears the associated interrupt condition in the device (714). Interrupts are then disabled (716), and interrupts are redirected from the device to interrupt handlers in the OS (718). Next, the hardware is configured not to trap the OS's I/O accesses to the device, and to let them complete directly to the hardware (720). Since the VMM will no longer trap those I/O operations, completing this step stops I/O emulation and state machine processing. Finally, interrupts are re-enabled (722). From then on, the OS performs I/Os directly to the device, and receives its interrupts, without the intervention of the VMM. An alternative to redirecting the device's interrupts to handlers in the OS is for the handlers in the VMM to quickly invoke the OS's handlers, without extensive processing.

[0050] The VMM can set a limit on the amount of time the VMM will wait for the state machine to reach the start state (708). If the start state is not reached in that time, corrective steps may be taken such as resetting the I/O device.

[0051] During both virtualization and devirtualization, the VMM might save the mode of the I/O device. I/O devices often have different modes of operation. Consider a UART, which handles serial communication. Typical UARTs can operate in a mode in which characters are seven bits long, with even parity, and where the UART generates an interrupt when it receives a new character. Often, those same UARTs can also operate in a mode in which characters are eight bits long, with odd parity, and where the UART doesn't interrupt the CPU when a new character arrives (i.e. it is in "polling" mode). The particular mode of operation for the device is often configured by its driver.

[0052] When virtualizing or devirtualizing an I/O device at runtime, the mode of operation employed by a device's driver in the OS should match the mode employed by the driver in the VMM. The VMM device driver can either be designed to work in the same mode as the OS driver, or the VMM driver may query the device to learn its mode of operation and then adopt an appropriate mode of operation for itself.

[0053] Alternately, if the VMM driver prefers to operate the device in a different mode from the mode employed by the OS's driver, it can query the mode of the I/O device during runtime virtualization, and save the mode in memory. After saving the mode, the VMM driver can reconfigure the device to the VMM's preferred mode of operation. Later, at devirtualization time, the VMM can restore the device to the OS's preferred mode of operation. For devices with multiple start states in their state machines, the VMM may choose which start state to employ when commencing emulation (during step 218 of Figure 2) based on the configured mode of the I/O device.

[0054] Thus disclosed is a VMM that can reduce unnecessary overhead. An operating system need not be modified to run on the VMM or designed for that purpose, but instead may be a commodity operating system without special support for running in a virtual machine (such as Microsoft Windows, Linux, Tru64 Unix, etc.). Similarly, the raw hardware layer need not provide special support for the VMM. The VMM can work on commodity hardware not having features that help or accelerate virtualization (such as Intel x86, Intel Itanium, and HP Alpha processors).

[0055] The runtime virtualization and devirtualization were described above for a single I/O device. However, the runtime virtualization and devirtualization can be performed on more than one I/O device at a time.

[0056] The present invention is not limited to the specific embodiments described and illustrated above. Instead, the present invention is construed according to the claims that follow.